Department of Computer Science

Factorio Blueprint Layout Optimisation

Thomas Burniston

Supervised by Peter Nightingale

01 May 2024

Submitted in part for the fulfilment of the degree of Computer Science with AI (With Year in Industry) MEng at University of York.

TABLE OF CONTENTS

Executive Summary **iii**

1 Introduction **1**

2 Literature Review **4**

2.1 Prior Solutions 4

2.2 Local Search 6

2.3 Conveyor Routing 8

2.4 Multi-Agent Pathfinding 9

3 Methodology and Implementation **13**

3.1 Problem Parsing 13

3.1.1 Information Extraction 13

3.1.2 Run Configurations 14

3.2 Layout Optimisation 15

3.2.1 Local Search 16

3.2.2 Conflict Based Pathfinding 18

3.2.3 Low Level Pathfinding 23

4 Results and Discussion **26**

4.1 Results 26

4.2 Future Work 29

4.3 Summary 30

Appendix A: Problem Run Configurations **31**

Bibliography **32**

TABLE OF FIGURES

Figure 1: World generation [2]. 1
Figure 2: Example factory [1]. 1
Figure 3: Search stages overview. 13
Figure 4: Example local search state. 17
Figure 5: Example local search state with item endpoints. 19
Figure 6: Example suitability evaluation. 21
Figure 7: Processing of endpoints to paths, using Figure 5. 22
Figure 8: 3 example problems for evaluation. 26
Figure 9: Local search algorithm evaluation graph. 26
Figure 10: Full solver evaluation graph. 27
Figure 11: Problem 2 evaluation graphs. 28
Figure 12, 12a-c: Problem 2 definition and examples. 28
Figures 13a, 13b: Highlighted problems with path resolution. 28

TABLE OF TABLES

Table 1: Local search algorithm evaluation results. 27

TABLE OF APPENDICES

Appendix A: Problem Run Configurations 31

Executive Summary

Factorio is a 2D top-down sandbox game made by Wube Studios [1]. The player is placed into a procedurally generated [2] world where they must extract raw materials, process them into refined components, and build up automated factories. The player has access to various placeable machines to automatically transport, refine, and combine items. As you progress through the game the amount of resources required increases, and as such you need to grow your factory larger and larger. The key tools the player has access to are assemblers, inserters, and conveyors. Assemblers produce items based on their given recipe, but require inserters placed besides them to move items in and out. Conveyors transport items placed onto them and are used for routing between assemblers. Factorio provides functionality for user defined blueprints which allows the player to save a section of their factory to be reused later or serialized and shared externally. These have various uses, however importantly they can be used for saving optimized sub-sections of your factory.

This paper tackles the problem of optimizing the layout of a blueprint given a specific problem definition with the blueprint dimensions, input item positions and rates, and the requested output item positions. A primary goal within Factorio is that of a wholly efficient factory; if a user is capable of optimizing re-usable blueprints they could theoretically use these as components which can be slotted together. This specific layout optimization problem has been previously tackled by S. Patterson et al. [3], who reduced the problem to a Constraints Optimization Problem (COP). They divided it into 3 models with individual goals, and pass information between the stages, moving back and forth based on each stage's success. They found their model overall was successful and was capable of finding solutions for blueprints with around 100 tiles within a few minutes. Some key issues they noted due to the use of Constraints Programming (CP) was that they were not able to use a single unified model nor were they able to model continuous item flow rates, due to the search space growing too large. However, with 3 separate models passing information between each was limited.

The approach taken in my paper aims to evaluate the effectiveness of using methods of local search and multi-agent pathfinding compositely. Multi-agent pathfinding is a method of solving multiple separate pathfinding problems while preventing conflicts, which I used for item routing. A key overall intention was to avoid the problems posed by the CP approach, specifically the incapability of modelling continuous flow rates of items due to the strictness of their framework.

My solution uses an initial pre-processing stage similar in function to the first model from S. Patterson et al., which given a problem definition extracts a series of possible run configurations. These each define amount of assemblers for the final output item, as well as the amount of assemblers and inserters for each component item. These are found deterministically for a given problem definition with a search through the given item recipes and limited by the rates of the problems input items. The primary layout optimization utilises a multi-level search, in which I use Simulated Annealing (SA) [4] as an upper level local search, with a Conflict Based Search (CBS) [5] as part of the fitness calculation for each state. The SA algorithm's primary purpose is the placement of the assemblers and inserters within the blueprint, and the fitness for each state is calculated in 2 parts. First, the overlap penalty, is calculated by summing the number of overlaps between the placed objects, which when used effectively allows the local search to solve a 2D placement problem. If a state is evaluated to have no overlap penalty, the second part of the fitness uses CBS for multi-agent pathfinding of the conveyors to route the items between the assembler's input and output inserters. The items being routed may have differing amounts of sources and destinations, and therefore my paper introduced a method for handling these disjoint amounts within the context of the conflict-based search. I used A* search [6] as the bottom level pathfinding. Each A* state encodes a position, a type, and a direction. To work alongside the CBS a custom definition of a conflict was required, which I modelled based on how the states would interact in-game.

Evaluation found the implementation was successful in finding solutions but found an oversight in the chosen model when resolving paths that connect onto other paths. Further optimizations on the method as a whole may involve hashing CT nodes in CBS to prevent recalculation, using the conflict information gathered in the CBS to break ties between A* nodes, and considering conflicts between more than 2 paths. More drastic optimizations may involve adding intelligence to the routing of the A* pathfinding to consider paths which will route connect onto current path. My work expanded on the model of the game used by the CP solution [7] through introduction of underground belts, however the problem still has space for further work and improvements. There is no consideration for electricity distribution, nor does it consider the 2 separate lanes of the conveyors. Furthermore, no consideration is taken in this solution for conveyor splitters and mergers, which would allow precise item flow rate management. Overall, my methodology and implementation of a composition of CBS and SA was successful in finding solutions for the Factorio blueprint layout optimization problem. There were no legal, moral, or ethical concerns.

1 Introduction

Factorio is a sandbox, top-down, 2D video game produced by Wube Software [8]. The game takes place on an infinite square grid, procedurally generated with ore veins, oceans, forests, deserts, enemies, and more. When you start you aim to pick an area rich with resources, begin extracting raw materials from the ground, and start refining them by hand. The main goal of Factorio is to scale this process of material extraction and synthesis with automation, through placeable buildings that can mine, craft, and transport items.

As you progress through the game you need to make more and more complex materials and consequently build bigger and bigger factories. To advance and unlock further technology you spend science research packs, which come as a set of increasingly difficult and more resource intensive goals. This increase in demand of raw resources forces the player to optimise and scale their factory and infrastructure. The materials in the world are finite, and therefore this forces you to also spread out laterally across the world to access more rich resource deposits, which the game accommodates with functionality for train networks, with capacity for automation with scheduling and signals.

The native inhabitants fight back against this expansion and attack your factories and the player, requiring you to setup automated defense, as well as offensive vehicles and weaponry. This assault increases as you produce increased pollution with larger factories, requiring the players defense to grow alongside their production.

Figure 1: World generation [2].

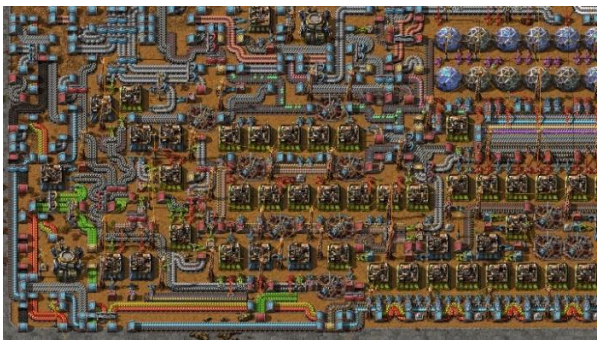

Figure 2: Example factory [1].

The factories you build consist of buildings that can manufacture (e.g. miners, smelters, assembler, pumps, refineries) and methods of routing (e.g. conveyor belts, conveyor splitters and mergers, inserters, liquid pipes). Minimally, the inserters and assemblers are your key tools for production; assemblers produce items following the items recipe and require items to be transported in and out with inserters.

Inserters move items across themselves from one side to the other, optionally taking from or placing into other machines.

As your factory grows, the logistical challenge of routing conveyors and designing scalable and efficient layouts becomes challenging. Routing a few items between a few assemblers can be trivial, but as you scale and your throughput increases, you need multiple belts transporting many distinct items and distributing these across a large and complex layout of assemblers. The challenges the player faces in the game manifests as series of overlapping non-trivial problems, e.g. packing of machines within a layout for space efficiency and scalability, routing of non-overlapping conveyors each with separate items and flow rates, scheduling trains with automatic routes, efficient defense of your factories from attackers, and managing the scaling throughput of raw items / infrastructure to name a few. The complexity and interest of the game comes from the intersection and handling of these problems, and the nuance that brings.

A range of these problems have been formally considered and explored. A specific problem of balancing item flow rate across multiple conveyors, i.e. keeping a series of parallel belts equal in the amount of items per second they are transporting, poses a contained problem that Leue [9] tackled. They posed Petri Nets with Linear Temporal logic as a method to verify solutions. Separately, Reid et al. [10] compared multiple methods of routing a single conveyor belt within the game world, using a novel modification to the game to expose a virtual interface, allowing their program to simulate and verify their solution directly.

A key mechanism Factorio provides is functionality for user made blueprints. The user may take a snapshot of an area of their factory that they can store and reuse in game or serialize to a code that they can share externally. This becomes vital for later game expansion where efficient sub-sections of your factories effectively become large re-usable components, allowing the player to speed up trivial tasks and reduce mental overhead. While there is an upper limit of 10000x10000 tiles for the size of the blueprint, the blueprints the player will make are generally less than 100x100, depending on the use case. By providing the player this functionality it implicitly poses the challenge of the optimal layout of a blueprint given specific use cases and scenarios. For example, you may aim for the most efficient use of space to do a single defined task, the most efficient rate of processing within a set defined area, or even optimize for a tile-able layout.

This paper focuses on the problem of how to optimise a blueprint layout, minimising the cost of the item routing and maximising the final item output. This is given a set blueprint width and height, the set input item’s rates and locations, and a desired output item rate and location.

I model a subset of the game's features. I abstract conveyors to be a single flow of items; conveyors in Factorio can have different items on each of their two sides with completely separate flow rates. For simplicity of modelling, I combine these flow rates to a single lane. Furthermore, I do not consider electrical requirements for buildings, and nor do I attempt placement or consideration for electrical poles. Both inserters and assemblers require an electrical pole within a set radius supplying power, and the placement of the poles adds increased complexity to the search. The primary intentions and objectives of this paper is to explore various methods to solve this problem and achieve the following objectives:

- Apply Local Search to find the optimal layout of blueprint.
- Apply a Conflict Based Search to Multi-Agent Pathfinding for optimally routing conveyors between defined points.
- Investigate applicability of a novel composition of these algorithms for blueprint optimisation.

Local search is a method of search through an induced state space that specifically only considers the neighbourhoods of each state. I am intending to be applying this to the placement of the assemblers and inserters. Then for item conveyor routing, I am solving them as a Multi-Agent Pathfinding (MAPF) problem using a Conflict Based Search (CBS). The problem at hand is of routing multiple conveyors between sources and destinations, without overlapping, adhering to the abstract model of the game I am simulating. This composition of methods for this problem requires additional novel processing in between to deal with unique domain specific situations. When routing items there may be a disjoint number of source and destination positions that each require pairing and satisfying. I will first explore and compare these methods as well as past literature on Factorio, describe my final methodology and reasoning in depth, then summarize and document my results against a set of examples.

2 Literature Review

2.1 Prior Solutions

S Patterson et al. [3] put forward an initial solution for optimizing blueprint layouts in Factorio through *Constraints Programming (CP).* Constraints programming is a methodology used for solving problems through a definition as a set of variables, domains, and constraints, and through a process of satisfying these constraints by optimally assigning values to the variables within their respective domains. This paper specifically defines the problem as a *Constraint Optimization Problem (COP)*, which aims to minimize / maximize the additionally defined objective function. In this case, they defined optimality of a blueprint as maximising production rate, and minimising cost.

During preliminary analysis, they found CP imposed limitations on their model which meant modelling continuous values was unfeasible. CP requires domains to be explicitly defined ranges, meaning even a compromise of integer values (1..450) for potentially decimal item flow rate was too broad to be applied for every tile of the blueprint, and caused state space to explode in size, especially in a single unified COP model which covered the entire problem. Along with other preliminary testing, this led them arrive at the decision to divide the problem into a multi-stage COP model, in which they optimise 3 separate models.

Their first stage was the recipe stage. Producable items in the game have a defined recipe that specify the input items and their quantities, the output item quantity, and the duration required for it to complete. As parameters for their first stage of processing, they defined the recipes as they are given in Factorio with all of the above parameters for the relevant items to the problem. Then, using the problem defined expected output item and the defined input items, they solved for the number of assemblers and inserters needed for maximum flow rate. That is, how many assemblers and inserters of each item are needed given the maximum possible output item rate as calculated from the input item rates. This provided a baseline upper limit for the next stage, as well as with careful iteration a set of possible arrangements of counts of assemblers, in case the maximum is not valid. Validation of a specific count of assemblers was identified as a formal problem of *Knapsack* and *Bin Packing* [11], which they go onto solve in stage 2.

Knapsack and *Bin Packing* are both problems that take a series of items with weights / dimensions / values and aim to fit them into a set sized area. These are both well researched and applied within CP, and therefore were able to be directly applied. Care is taken to allow this stage to return to the previous stage and request the next set of counts for assemblers and inserters if the space checking failed. At this point

they move onto the primary stage 3, which tackles the key problem of placing assemblers and routing of items.

For stage 3 they proposed *Multi Agent Pathfinding algorithms (MAPF)* [12], and specifically *non-crossing CP MAPF* [7], as a potential solution. MAPF is the problem of pathfinding multiple agents over a state space at the same time, with each agent having different source and target states. 2 agents cannot be at the same state, at the same time step. Non crossing MAPF is a subset of this problem such that agents cannot cross at any point in time. They go on to discuss how Yu and LaValle [12] reduce non crossing MAPF to a network flow problem. They point out a range of incompatibilities with applying it to their exact routing problem, such as multiple source nodes with the same target node (and vice versa), as well as groups of routes that can and cannot overlap. While they state that existing methods cannot be directly applied within this context, a lot of their fundamental ideas are applicable. The idea of using aspects of various MAPF methods within this sub-stage was particularly insightful, and I further discuss my chosen conveyor routing methodology in **section 2.3**. They do not continue on to use MAPF in their solution.

They pose a trade-off for flow rate optimality during their discussion of item path splitting and merging across multiple conveyors during stage 3. With a final solution, each input inserter on each assembler requires a certain flow rate of their specific item, and the output inserter on each produces a set amount. It is intuitive that this information would be considered during pathfinding, however they choose to only consider flow rate during the inserter and assembler counting in stage 1. A key issue this can cause is of conveyors being starved by inserters upstream taking more than the calculated amount. They note that due to the expectedly tightly packed nature of generated solutions, it is likely not an issue, however it is possible the solutions are not optimal.

Their multistage approach needs to serialize and deserialize information between stages, as well as have a capacity to step between and repeat stages when attempts fail. During evaluation of their final solution, they note this is detrimental to the overall solvers performance and limited the possible information flow.

Furthermore, due to an aim reduce complexity of the solution, they limited the functionality from the game they considered and modelled. Amongst other things, this included underground belts and electricity. Within Factorio underground belts provide the user means of transporting items underneath other buildings and belts, specifically in a straight line. This is a key component for more compact and efficient blueprint layouts. Their model choosing to not include underground

belts means solutions produced by their solver may be suboptimal in compactness. Most machines within Factorio require an electrical supply to function. This is provided with electric poles, which have a radius around them in which buildings are powered, and a separate larger radius in which they connect to other electric poles. It is important to consider the placement of the electric poles especially when aiming for compact or scalable layouts. The solutions their method generates do not consider this, meaning it is possible to produce a layout too compact which cannot be supplied with electricity due to no space for the poles. To be able to consider electric pole placement would increase the complexity of their item routing stage, therefore their decision to omit this was in aid of the balance of fidelity and performance.

A further design decision they made was to simplify the conveyor model from two separate sides to a single lane. Due to this it meant their routing stage had a more simplified view of item flow rate, which in turn meant it generally would produce sub-optimal results compared to what is possible. Furthermore, the model did not fully consider cases where an inserter might be starved of item due to other inserters upstream consuming more than the required number of items. This was in part due to their omission of flow rate information during item routing stages. This means their predicted final item output rate was not guaranteed to match a simulated and measured amount; it was deemed this was a necessary compromise for a *tractable* model, and to ensure the model could solve even trivial instances, as increases in fidelity of item flow rate requires increased model complexity. In the circumstance of a compact system, assemblers internal item buffers would fill up over time and the inserters would eventually stop pulling extra items, meaning it would eventually even out and stabilise. The solutions created by their method were observed to be correct when replicated in game, such that they would manage to produce the correct items with an additional manual step of placement of electric poles.

2.2 Local Search

The problem of solving blueprint layouts, as found above by S Patterson et al., invokes an exceptionally large search space. The blueprint can be of an arbitrary size, with an arbitrary number of different items, each with multiple assemblers, multiple source and destination inserters, and multiple belts transporting them each with individual flow rates.

S. Russel and P. Norvig describe local search [4], a form of search algorithm which relax strict requirements and instead explore and evaluate states only in a local neighbourhood, as opposed to a systematic search over a full search space. Local search does not

expect full knowledge of all possible states and instead considers and evaluates a single state at a time, then progressing to a neighbouring state based on the specific algorithm. This means that it can be applied to problems with excessively large state spaces. The problem at hand as described above is apt for application of local search.

An initial algorithm described by S. Russel and R. Norvig is *Hill Climbing* [4] (**p122**). From each state, always moves towards the neighbour of increasing value. If none exist, then a local optimum has been found and is returned as the solution. While this algorithm is **complete**, i.e. it is guaranteed to find a solution, it is not **optimal**, in that it holds no guarantee to find the best. Due to being *greedy* (directly goes to best neighbour with no consideration) it is prone to a range of issues, including getting stuck in local maxima (a locally maximal peak that cannot be escaped), ridges (a sequence of local maxima), and plateaux (a flat area of state-space).

As an improvement to Hill Climbing, *Simulated Annealing* [4] (**p125**) is proposed. At each state, instead pick a random neighbour; If it is an improvement always move to it, however if it is worse move based on a calculated chance as a function of the delta of the neighbours' fitness to the current state, and the algorithms current *temperature.* The temperature is initially high and gradually reduces to zero, simulating metallurgic annealing in real life. This process aims to solve the closed view problems greedy algorithms have by allowing the algorithm to move to lower fitness states, in hopes of finding global optima instead of getting stuck in local optima.

Further improvements come in the form of maintaining multiple agents. *Beam Search* [4] (**p125**) proposes maintaining k solutions, where at each step useful information is shared between them and helps inform chosen successors This means that successors are chosen from the pool of neighbours of all current states. *Stochastic Beam Search* improves on this by making the choice of successor as a function of its value, to prevent the states clumping in the state space. *Genetic Algorithms* [4] (**p126**) take this idea and apply a biological parallel. The k states are deemed individuals of a population, with each subsequent generation being produced by combination of 2 parent individuals, with included random mutation. The solution representation must support the concepts of crossover and mutation for this to be applicable.

As initially described, the state space of the blueprint problem has the potential to be huge. S Patterson et al. [3] solve this by splitting their model into 3 stages each with separate search spaces. Later in **section 3.2** I will show some form of local search is applicable to a sub-section of this large optimization problem.

2.3 Conveyor Routing

Reid et al. [10] proposed and compared a variety of methods for routing a single conveyor on a grid from a source point to a target point. Furthermore, they implemented a direct interface to a running instance of the game, *Factorio Optimizer Interface* (FOI), for use with verifying solutions. They explore 3 main methods of routing conveyors: *Parallel Simulated Annealing*, *Genetic Programming*, and *Reinforcement Learning*.

Simulated Annealing (SA) is a form of a local search as described in **section 2.2**, that moves between neighbourhoods of states based on their fitness, with an additional temperature parameter that dictates an amount of randomness in accepting lesser fit states. In this paper they describe how *Parallel Simulated Annealing (PSA)* can be applied to the conveyor routing problem. PSA is a revised form of SA that handles multiple solutions in each iteration. That is, it stores a set of solutions, calculates a set of fitnesses, and iterates them all in parallel using a shared temperature at each iteration. The purpose behind their choice of PSA was that they noted proven capability of similar algorithms [13] when solving problems with larger search spaces, and serial SA's specific inefficiency with this problem.

Genetic Programming (GP) is a method of solving optimization problems that can be digitally represented in a way that can be evaluated, mutated, and crossed over, similarly to Genetic Algorithms as described in **section 2.2**. The key difference is the additional mutation and crossover of the *representation* through the algorithm. They propose a novel variant, qGP, in which the genome is a sequence of strings and integers. They interpret the genome by parsing to operators and operands, in which an operator connects a set of operands and maps to another set of operands. In this case, the operators and operands refer to placement and machines.

Reinforcement learning (RL) is a category of machine learning that simulates agents adapting over time; in each discrete time step they observe, take an action, and receives feedback. In this paper, they evolve their agents using a Genetic Algorithm, where in each generation they evaluate the solutions against 5 unique scenarios to ensure the learnt behaviour is generalised and not overfit. The bottom 50% of a generation is culled and replaced by crossing over parents from the surviving half. Solutions are given a maximum lifetime of 20 generations, or 100 scenarios, to ensure single dominating solutions are not kept alive. A RL agent is defined with a partial world view, e.g. at each time step this agent is at a specific position and has a 3x3 matrix of cells it can view, can only move in 1 of 4 directions, and can place a transport belt in 1 of 4 directions.

Overall, the methods they implemented were able to find solutions for up to 12x12 blueprint plots, including obstacles. qGP consistently produced the best solutions, they speculate due to its problem specific genome representation. RLs evaluation computation cost caused it not to be able to be evaluated for any problem other than 3x3, in which even then it was not able to compete with the others. It was noted however that it was capable of generalization, as a key benefit.

Reid et al. explore a range of valid methods for routing conveyors, propose a novel method of verification, and give useful insight on dealing with and producing a solution for single belt problems in a large search space. This problem they tackle is adjacent to the wider problem of multi-agent conveyor routing I am intending to solve, and the insights into novelty of the representation is useful, however their in-game verification method is not a primary concern.

2.4 Multi-Agent Pathfinding

S. Patterson et Al [3] proposed using MAPF within their CP model [7], drew links to Network Flow [12], as well as solving the problem using this paradigm [14]. While they did not continue down this path of enquiry due to it not being directly applicable within their constraints programming context (as previously outlined), more literature and insight can be explored in this area.

A conflict-based search for MAPF was proposed by Sharon et al. [5]. They start by defining *coupled* and *decoupled* MAPF solutions. A decoupled solution considers an ordering of their multiple agents and solves their individual pathfinding problems in this order. With each path found, the world is updated to include this path for subsequent agents to consider and avoid. These types of methods typically are sub-optimal, and completeness is not guaranteed, however they do decompose the problem into a series of independent / loosely related problems on individual agents. This allows for more freedom with what algorithm is used for the low-level search, including standard greedy pathfinding algorithms. On the other hand, coupled MAPF approaches are generally *complete*, with each state containing information about every agent at their current timestep, and the pathfinding algorithm working across these composite states at a higher level. This means the search space for a coupled algorithm grows exponentially with the number of agents, which induces significant computational expense. To avoid the issues of each, Sharon et al. suggest CBS as an intermediary solution.

Sharon et al. first point towards work by D. Silver [15], who describes various decoupled MAPF approaches and applicable optimisations there-in. The general approach involves taking an ordering of agents and solving each individually with the A* algorithm [6], and then using

some method (of which he compares a variety) of solving conflicts. A* is described by S. Russel and R. Norvig [6], as a best-first search over a state space by evaluating each node against its cost to the start node, and a heuristic to the target node. This method is both complete and optimal, given an appropriate heuristic function. D. Silver starts with an initial solution of *Local Repair A* (LRA*),* where the agents each perform pathfinding separately with A*, then upon a collision when tracing back through recompute the rest of their path. They also maintain an agitation level which determines a level of random movement based on the running number of collisions, with the aim to prevent agents getting stuck in problematic areas. D. Silver's final solution in his paper, *Hierarchical Cooperative A* (HCA*)*, employs a range of optimisation over the initial simple algorithm. HCA* maintains a reservation table of agent positions that is updated after each path is found. The pathfinding then considers entries in this table as blocked states that cannot be used during their pathfinding. The *hierarchical* term in HCA* refers to their choice of heuristic, where the heuristic of each node to the target location is that of a more abstract view of the world, specifically one ignoring time, the other agents, and the reservation table. This lower-level A* search used the Manhattan distance as the heuristic. While this was effective, it was very computationally expensive.

A key optimisation is that of *Reverse Resumable A* (RRA*)*, in which the A* heuristic search is instead performed from the target node to the agent's current node, in such a manner that the information of all other states can be cached and re-used later. Using this, HCA* does not need to be recompute the heuristic from every node from scratch. If a node is reached that does not have any heuristic pathfinding information (as could happen after a conflict) the A* heuristic search can be *resumed* until that node is included in the cached heuristic lookup table.

A final optimisation they introduce is that of windowing. An agent will find a partial HCA* route to it's a destination (with a limited depth), and then begin traversing. After a set distance (e.g. halfway through), they can recompute and shift the window forward. Notably the heuristic basic A* search is not limited. Windowing allows them to continue the pathfinding after an agent has arrived at their target location, therefore allowing them to move out the way of others. Furthermore, by staggering agents' windows movement / recalculation they can smoothly spread the computation cost over the entire search, rather than a large initial computation.

Overall, they found HCA* and WHCA* (Windowed HCA*) did outperform *LRA** drastically on a variety of problems, in the metrics of cycles taken to complete, percentage of agents which successfully complete, and average path length, but do incur a significant overhead

of initial computation time. As per the expectation with a non-coupled algorithm, these were not complete and not guaranteed to find a solution for all agents.

Sharon et al. describe and explore a coupled approach [5], as a baseline for a complete solution. Each state contains information on how to place each of k agents into the V vertices without conflicts, and the neighbourhood is defined by valid movements of all agents that do not produce conflicts. A* can then be directly applied to this, with the heuristic as the sum of individual costs (SIC) over the agents, which in this case is a pre-calculated value stored in a look-up table of A* from each point for an individual agent. There are 3 existing optimisations that are described that can be applied to this problem and that are used in their final combination approach.

Independence Detection (ID) defines groups of agents as agents whose solutions do not cause conflicts. Applying ID to this coupled search involves initially placing each agent into their own group and solving separately. Then, finding pairs of groups that do have conflicts and merging and solving them, using some optimal MAPF solver. The final solution is found when no more pairs groups have conflicts. When solving a group there may be multiple optimal solutions. A Conflict Avoidance Table (CAT) keeps track of agents positions at each time *across* ID groups and can be used as a heuristic for distinguishing nodes with equal fitness, by prioritising nodes with fewer entries in the CAT. Operator Decomposition (OD) as a further optimisation describes intermediate search states, where an operator is performed on a single agent, and checked for validity. This can allow you to prune misleading states early without checking a state completely.

Sharon et al. introduce a new method of optimisation, with an ICTS. This divides the problem into 2 levels; A high level which searches through an Increasing Conflict Tree (ICT), whose nodes are defined with a vector of costs, each related to an agent. At the low level it performs a goal test on an ICT node and checks to see if there exists a combination of single-agent paths with their designated costs, or to verify if such a solution does not exist. They found this approach did outperform A* in cases where $\Delta < k$, where Δ is the difference between the SIC of the initial state and the optimal conflict free solution, and k is the number of agents. An optimisation denoted ICTS+3E uses information about groups of 3 agents within their ICT approach.

The final coupled / decoupled combination approach proposed by Sharon et al., *Conflict Based Search (CBS)*, is formulated with the intention of maintaining the optimality of a coupled approach, while still ensuring the pathfinding search space only considers single agents, as with the decoupled approaches. Their proposed method splits the

problem into 2 levels as with the coupled approach, a high level with a binary *Constraints Tree (CT)* and a low level that performs path finding.

Each node in the high-level CT is defined by as set of constraints (3-tuples of positions agents cannot be at a specific time step), a solution (a set of paths, 1 for each agent, that are *consistent* with the constraints of that agent), and the total cost (SIC of all agents). A goal node in this CT state space is one where the solution is *consistent* and *valid*, that is all the paths are consistent with their individual constraints, and there are no conflicts between agents. A best first search is then performed over the CT, using a CAT as defined before to solve ties between nodes. The low level is invoked at each CT node for each agent, considering that agents' constraints. This will find the shortest *consistent* path for that agent, one that adheres to the provided constraints. With a consistent path for each agent, the top-level CT node validates all paths against each other step by step, stopping at the first conflict. If a conflict is not found then this is a goal node and returned, however otherwise the conflict procedure is enacted. The CT is split into 2 branches, each one with an additional constraint for the conflicting position and time step, with each of the 2 branches defining this constraint for each of the 2 agents involved. Considering optimization, they found that they only needed to recompute the path of the 1 agent affected at each branch in the CT tree. And consequently, only store the new path of the affected agent in each new node.

Their paper concludes with a comparison between numerous methods, namely A*+OD, CBS, and ICTS+3E, across a range of scenarios. In their first scenario of a 4-connected 8x8 grid, they use ID in a preprocessing stage to find the largest group of conflicting agents to benchmark against. CBS consistently outperformed A*+OD, and for > 9 agent also outperformed ICTS+3E. When tested against a set of 3 maps from Dragon Age: Origins, they conclude that CBS still consistently outperforms A* + OD, but there isn't a definite global best. CBS runs best in situations with bottlenecks, but worse in wide open spaces. Within the case of Factorio belt optimisation, the expected scenario involves grids of size ranging 5x5 to 20x20, with very tightly packed areas, and 10+ agents. As per their final conclusions, CBS seems the most apt for this situation. However, the blueprint problem does not require timesteps to be involved and as such CBS cannot be directly applied and careful consideration will be required.

3 Methodology and Implementation

The intention of my approach, as outlined by the objectives defined in **section 1** is that of exploring the applicability of different methods of local search and multi-agent pathfinding to the problem of blueprint layout optimisation in Factorio. A problem is defined as follows: Given a blueprint with a specific width and height, as well as locations and rates for input items, and a requested location for an output item, find the layout of assemblers, conveyors, and inserters to produce the output item, maximising the flow rate, and minimising the cost.

The work done by S Patterson et al. [3] acted as a basis for my choice of the overall layout of my solution as a multi-stage approach, in which I initially extract information from the given problem definition, then perform an informed optimisation of the layout. The problem definition parsing also proposed by S Patterson et al. was directly applicable and described is in **section 3.1**. The layout is optimised with a multi-level search which entails a primary top-level search that utilises another search to evaluate the fitness of each state. More specifically, I use Simulated Annealing to optimise the layout and placement of the assemblers and inserters, then a modified CBS for MAPF for pathfinding the required paths and contributing towards fitness.

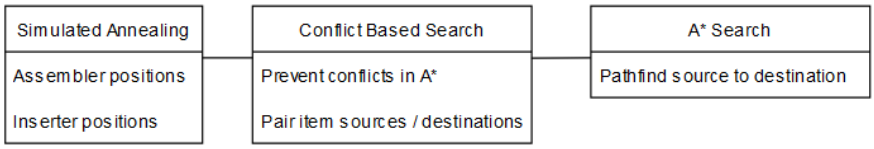

Figure 3: Search stages overview.

3.1 Problem Parsing

3.1.1 Information Extraction

When tasked with a specific problem to solve there is a set of information expected to be passed into the solver. This includes the blueprint width and height, information for each item input and single item output, and finally the recipes that the solver is allowed to use. The expected blueprint output item is just a position without rate, whereas the input items to the are at a specific position and rate. The recipe for each item includes a quantity produced, time taken, and a list of ingredient items with their required quantities.

With this problem definition, step 1 is to begin extracting useful information. Firstly, I need to calculate all the *component* items that I will need to produce with assemblers. By using the given recipes, I trace backwards from the output item through the induced recipe tree in a depth-first traversal. While calculating this, I also track the required rate per second of each of these items needed relative to the

requirements of a single assembler of the output item. That is, given the recipe $rate * quantity$ of the output item (e.g. 1 assembler producing the item with full input item rate satisfaction), what rate per second is needed of each component item. This is done by considering the quantity of each ingredient and rate of each recipe as I traverse. For this basic depth-first traversal I first push the output item (and its final rate as above) onto a stack, then while this is not empty I pop the top item, update my local component item information, and if it has a recipe add each ingredient to the stack along with their relative rates.

More concretely, when I initialize the stack with the output item I push it with the produced rate by a single assembler as per its recipe: $recipes[outputItem].rate * recipes[outputItem].quantity$. Then as I am moving through the recipe tree, I push each ingredient onto the stack with its relative rate to the current item as $ingredientQuantity * \frac{currentItemRate}{recipes[currentItem].quantity}$. This takes care to consider the recipes output item quantity and input ingredient item quantity being different.

3.1.2 Run Configurations

Now information on the relevant items has been extracted from the Problem Definition, the next step is to determine the concrete counts of the assemblers and inserters that I am aiming to place. To represent this, I define a *run configuration* as a structure uniquely identified by the amount of output item assemblers and contains the corresponding number of assemblers for each component item as well as the number of output inserters and input inserters for each assembler. Each problem definition may have multiple possible run configurations due to the input item rates being large enough to support different multiples of output assemblers. The run configuration for each count of output assembler can be deterministically calculated. A run configuration is intended to only represent the count of each machine and is used as input to later stages which attempts to positions them.

The idea for constructing these is that for a given output assembler count, there is a calculatable minimum number of assemblers for each of the component items needed to satisfy all the relative rate requirements. The overall goal from this point onwards is to find the run configuration representing a *valid* solution that maximises our goal of output items per second, and therefore has the highest number of output item assemblers. The requirements for a *valid* run configuration are determined by the end user's strictness on the solution. At minimum, a run configuration should require that a layout is found by the next stage with no overlapping machines. It may be that you also want to require that all the items within the solution are routed successfully with a valid path with no conflicts.

The maximum possible supported configuration (not considering validity) can be found using the rate of the problem definition input items. Given the rate of each input item, as well as the relative rate of that item to the output item as found in stage 1, divide to find the maximum multiple of output item assemblers supported and round up. Once this maximum is found, a run configuration is produced for each integer count from the maximum down to 1. This set of configurations are *all* the possible concrete amounts of assemblers and inserters supported by the input rates. These are all calculated so that if the subsequent layout optimisation stage cannot find a valid solution for a given run configuration then I can iterate backwards until one is found.

After calculating these run configurations, I need to perform they physically fit within blueprint bounds, however this isn't a trivial task. S Patterson et al. [3] tackle this problem using 2D bin-packing [11] with the 3x3 assemblers shapes and ignoring inserter positions. In my solution I have opted for cheap bounds checks at this stage and perform a concrete 2D placement problem with the local search as described in **section 3.2**. This decision was made due to it being a straightforward addition in the local search state fitness evaluation, as opposed to an extensive separate check here. I also can check the placements of the inserters in that stage without much more effort. For this preliminary stage, I check whether the sum of space required by the assemblers and inserters fit within the maximum area of the $blueprintWidth * blueprintHeight$, minus 1 for each blueprint input item and output item. I take up 9 squares for an assembler, and 2 squares for each inserter to include both the inserter position and the position opposite the assembler which it transports items from / to. I stop this space check once I have found the maximum configuration that passes, as I know that each lower run configuration will also fit.

At this point I have a set of run configurations that are supported given the rates of the input items and the recipes, as well as a starting upper limit as determined by preliminary space check. This is enough information to begin the optimization of the layout.

3.2 Layout Optimisation

As previously mentioned, the search for this problem is split into 2 layers. Concretely this means there is a top-level simulated annealing local search whose goal is to optimise the placement of assemblers and inserters, and a conflict-based search for multi-agent pathfinding underneath which routes the items and contributes to the top level's fitness calculation. The local search also handles checking whether the current run configuration can fit in the blueprint space, effectively solving a 2D placement problem. The choice to split into 2 layers was primarily due to consideration of each algorithms best use cases. if SA was used to also consider the routing of conveyors and perform the

pathfinding as well as placement of assemblers and inserters, the search space would be enormous. Furthermore, designing a heuristic for this would be a difficult if not impossible task.

Coming into this stage I have a set of sorted run configurations and begin by performing the search over the first configuration with the maximum amount of assemblers. If no valid solution is found (based on the previously described validity requirements) then I can iterate down to the next run configuration. Depending on your strictness on validity, it is possible there are no run configurations that a valid layout can be found for. For example, if you require that all the items are fully routed it may be that none of the layouts for any run configuration can meet this requirement. Alternatively, if you instead relax this requirement and only require for some of the items to be routed, it is possible that the first found *valid* solution is not in fact the one that maximises the objective, i.e. that some other lower run configuration may actually be able to route more items and overall produce higher output items per second, despite having less output item assemblers. Instead, my chosen approach is to calculate the layout for all the run configurations, then take the one with the highest fitness. Validity can then be considered manually on each final solution.

3.2.1 Local Search

The local search stage takes in a single run configuration containing an amount of assemblers and inserters. The states of the local search encode concrete placements of each assembler for each item, as well as the position of the output and input inserters on each assembler. Each assembler is defined as having 12 side locations, defined clockwise starting from the lefthand side on the top edge, that the inserters can be placed in. The *neighbourhood* of each state is defined by either a movement of 1 assembler, or a movement of 1 of the inserters to another side on 1 of the assemblers.

- Move a given assembler 1 position N / E / S / W.
- Swap inserter from position i to j on a given assembler.

An assembler movement is only defined when each grid cell within its 3x3 footprint as well as all attached inserters positions are kept within the bounds of the blueprint (i.e. $0 \leq x \leq blueprintWidth, 0 \leq y \leq blueprintHeight$ for each x, y in assembler and inserters). Equally, an inserter can only be moved to another edge position if it is still within the blueprint bounds, and not taken up by another inserter.

With this defined state and neighbourhood, the next key aspect of the local search is an effective fitness evaluation. I calculate the fitness of a given state as sum of 2 parts, the *overlap penalty,* and the *pathfinding fitness*. The *pathfinding fitness* is calculated using the

conflict-based pathfinder as described in **section 3.2.2.** However, this is only calculated if the *overlap penalty* deems the state *valid*.

The *overlap penalty* is a metric calculated by counting the number of overlaps in the grid between objects. That is, for every assembler and inserters concrete placement in a local search state, count the number of times a position is occupied by more than 1 object. For example, if the only overlap in a state is 3 inserters in the same position, then the overlap penalty = 2. A *valid* state is one with no overlaps, and therefore overlap penalty = 0. Each assembler is represented by a 3x3 footprint, whereas an assembler is 1x2, oriented to cover its position and the position opposite its assembler. When calculating this overlap penalty, I also calculate the *blocked grid* and the *item endpoints* to be used by the CBS MAPF search. The *blocked grid* is a Boolean 2D nested array that indicates which cells within the blueprint grid are blocked for the pathfinding stage. The entries are set true for all positions in the 3x3 assembler footprints, and true only in the 1x1 position of the inserters. We do not consider the other position of the inserters blocked as we need to be able to path find through these positions. Each calculated *item endpoint* is a given item, position, required flow rate, and is either a source or destination. These are calculated by processing all the input / output inserters on all the assemblers, as well as the problem definition inputs / output items. These item endpoints act as a set of requirements for the pathfinding stage to satisfy.

The following *Figure 4* is a visual representation of what a state may look like, with greyed cells indicating assembler / inserter in the blocked grid, and coloured cells indicating the item endpoints.

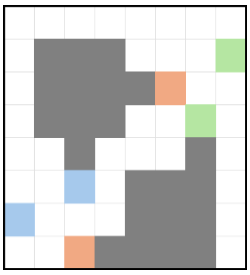

Figure 4: Example local search state.

As previously mentioned, the local search itself is an implementation of simulated annealing [4]. This algorithm was chosen to allow for the search to pass through less fit states to escape local minima, and hence be able to better explore the state space. For example, it may need to have to move an inserter to a blocked position and then move the assembler afterwards to unblock it to get to a higher fitness. The most fit state is kept track of throughout the search, and after a set number of iterations the best is returned. As opposed to a best first hill climber in which you always pick the neighbour with the highest fitness,

simulated annealing picks a random neighbour and decides to move based on an acceptance function of the fitnesses of each state, as well as of the current *temperature*. An initial temperature is defined as a meta parameter to the search, which is decreased towards 0 based on the temperature schedule. In my search I chose to a use a schedule with a cooling factor meta parameter, where each iteration temperature is updated as $t_{n+1} = t_n * (1 - c)$. Generally, I aimed for the acceptance function to allow states with less fitness through more often towards the start, and gradually become more stable on fit state towards the end. Decisions on specific meta parameters is described later in the results and evaluation in **section 4**.

3.2.2 Conflict Based Pathfinding

Once I find a valid local search state with a no overlap penalty, I need to route the items using the previously calculated blocked grid and item endpoints. This result of this is the *pathfinding fitness* which contributes to the overall local search state fitness. *Figure 4* previously showed a simple scenario with 3 sources and 3 destinations for the 3 relevant items in that problem. Complexity arises at this stage due to the possibility of an uneven number of sources and destinations for each item, which in turn gives us problems with directly applying MAPF. Furthermore, this problem does not need to consider a time component as the agents are permanent path placements and not moving agents, which also strays from the usual problem definition as per the literature [4]. Therefore, there is the following problem setup:

- 2D Boolean grid representing blocked coordinates.
- ≥ 1 source $(x, y, rate)$ for each component item.
- ≥ 1 destination $(x, y, rate)$ for each component item.
- Custom pathfinding states with no time component.

Even if there is equal number of sources and destinations for a given item, there is no guaranteed they can be matched up 1 to 1 and still satisfy each endpoints rate requirements. A real player of Factorio may use conveyor splitter / mergers to distribute items and balance rates. Pathfinding multiple agents, satisfying the rate requirements, supporting underground conveyors, *and* supporting conveyor splitters and mergers was deemed too large of a task to tackle at this stage. A complete solution may involve dynamically satisfying item destination requirements by splitting item from other calculated paths, and remerging left-over items back onto other existing paths, while also considering routing to multiple endpoints from each source. Doing this for every provided source and destination, as well as with conflict resolution across the multiple paths, inside every single local search state would likely take an excessive amount of computation time and be very complex to implement, making the upper-level local search take too long to be done in reasonable amount of time.

The solution therefore must have realistic expectations. Firstly, I accept that calculated path cannot completely satisfy the rate requirements of the sources and destinations, and only aim to ensure every source and destination has a path connected to it. Subsequently, I do not model conveyor splitters and mergers and do not intend to manage item flow rates with that level of precision. For working towards a solution to this problem, I can make the assertion that every item with at least 1 source will have at least 1 destination, and vice versa. This means that for each item there will be at least 1 source and destination that I can path between, and in turn which other disjoint endpoints (sources and destinations) can connect onto.

Before I can begin to choose a CBS MAPF algorithm, I need to consider how to extract concrete *path configurations* from a series of endpoints. A path configuration is a concrete definition of a source and destination for a given path, in which each end is either another path or an item endpoint. To explain, I will use the following example state, with the item endpoints highlighted in the diagram and enumerated as tuples of *{ item, rate, coordinate, isSource }:*

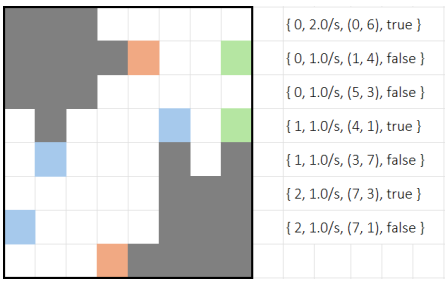

Figure 5: Example local search state with item endpoints.

While items 1 and 2 are simple cases (1 source, 1 destination), item 0 has 1 source and 2 destinations. To be able to handle these unbalanced item endpoints, it's intuitive to reason that you can initially calculate the path from the 1 source to 1 of the destinations, and then split the items off of this path with inserters and route to the other disjoint destination. When pathing from an already determined path to an endpoint (or vice versa) I can pick from any point beside the path that is unblocked, and adjacent to an above ground segment. I choose to prioritise the closest point to the target endpoint, as it is intuitive to see that this closest point will minimize the overall cost of that specific path. If the low-level pathfinding cannot find a path with the decided point on the path endpoint, it could be reasonable to consider one of the other possible adjacent points to the path, however I leave this as room for future improvement due to time and implementation complexity limitations.

The procedure for producing a set of path configurations from a list of possibly disjoint item endpoints is as follows. I define the *spare rate* of

a calculated path configuration initially as the available rate of the source endpoint minus the required rate of the destination endpoint. Conceptually, if positive, this is the left-over unused item flow rate, and if negative, it represents the extra rate needed to satisfy the destination endpoint. I group up and process the item endpoints separately for each item, sorting high to low by the required rate. While this list has endpoints to process, I take the first in the list (i.e. the most demanding item source or destination endpoint) and look to pair with the most suitable endpoint, be that another unprocessed compatible item endpoint or an already calculated path configuration. When finding the most suitable path endpoint to pair with I search through all the current unprocessed item endpoints that are compatible (e.g. source with destination) and *all* of the current items calculated path configurations. I use the following rules to decide:

- 1: Minimize $|source - dest|$, prioritise if $source - dest > 0$
    - **Destination**: Endpoint
- 2: Minimize $|source + dest|$, prioritise if $source + dest < 0$
    - **Destination**: Path

$source$ and $dest$ are the corresponding rates of the 2 path endpoints I am considering pairing. A path endpoint which satisfies the priority inequality of its given rule will always be picked over a path endpoint that does not, otherwise I minimise the value specified.

The choice of rule is based on whether the destination endpoint is another path or an item endpoint. Rule 1 with an item endpoint as the destination the priority inequality encodes the priority for the source to produce enough for the destination, and the optimized value encodes preference for as little excess as possible. With a path as the source this is considering the paths current spare rate, whereas with an item endpoint source this is the defined output rate of the inserter. However, with rule 2 with a path as the destination, we are instead looking to optimize for the path that needs the source item rate the most and hence the equations are slightly different. The priority inequality prefers paths that will have negative spare rate even after this path connection is made (and needs the items the most). The minimised value $|source + dest|$ encodes preference to match the paths requirement as close as possible, as above with minimal excess.

Consideration was needed for updating the spare rates of complex connections of multiple paths connecting to each other correctly. If I use a path as a source or destination, I need to update the spare rate of that path accordingly. If I was to then use the second path as a source or destination, the calculation gets complex. Instead, to track the spare rates of multiple paths, I track the spare rates for *path groups*, indexed by a unique ID. When I make a new path between 2 item endpoints, this ID is incremented, assigned to the path, and the

spare rate initialized ($source - destination$). When I add a new path that uses another existing path as an endpoint, it will instead be assigned the same path group ID as that path. Now whenever I update the spare rate of a path, I instead update the shared path groups total spare rate. *Figure 6* contains an example:

Item Endpoints	
0	Destination: 8.0/s
1	Source: 6.0/s
2	**Source: 2.0/s**
3	Source: 2.0/s
4	Destination: 1.5/s
5	Destination: 1.0/s

Current Paths	
0	1 -> 0 | Leftover: -2.0/s

Figure 6: Example suitability evaluation.

At the current state of the algorithm in *Figure 6,* I have already initially paired destination endpoint 0 with source endpoint 1, with the minimized value using rule 1 being $|6 - 8| = 2$. Note, none of the sources were able to satisfy the priority inequality (and therefore have enough rate for the destination) and therefore the best fit of 2 represents the extra rate needed to satisfy the destination. The spare rate value $-2.0/s$ next to the first path in *Figure 6* represents path group 0's spare rate. Continuing the algorithm, I consider the source item endpoint 2 (as outlined in orange) and need to find the most suitable destination. In this case it will find the current existing path to be the most suitable destination using rule 2, finding $|2 + (-2)| = 0$ to be the minimum, and in fact also satisfy the priority inequality. Conceptually this is showing the source rate of 2 satisfying the 2 missing required rate for the path. For example, if you plug in the values for the other destinations, rule 1 with endpoint 4 at $|2 - 1| = 1$ and rule 1 for endpoint 5 at $|2 - 1.5| = 0.5$, they are both higher values representing more excess rate, despite all being prioritised. This specific setup finishes with paths *(1->0 | pg0), (2->p0 | pg0), (3->4 | pg1), (p1->5 | pg1)*. Path group 0 (pg0) will have 0 spare item rate, whereas path group 1 (pg1) will have -0.5 leftover.

When performing this algorithm, there is a guarantee for each item endpoint that there will always be another possible endpoint to pair with. I know there is always at least 1 source and 1 destination for each item, and therefore all subsequent disjoint sources or destinations can be paired with this path if necessary. There is however no guarantee that there is always free space besides a path, and therefore it is possible, albeit uncommon, that a path configuration cannot be resolved if it connects to another path, due to no available spaces. This may be due to other machines blocking the spaces or depending on the MAPF algorithm, the current constraints. I will explain later how I

deal with this when looking at specifics. There are also potential future improvements here to produce additional paths *between* paths with positive / negative spare item rate to improve evaluated fitness, however this was not implemented.

I further apply this algorithm to the example early from *Figure 5* In *Figure 7,* with endpoint tuples of *{ item, rate, coordinate, isSource }*:

Ordered Ends	#0
{ 0, 2.0/s, (0, 6), true }: 0	
{ 0, 1.0/s, (1, 4), false }: 0	
{ 0, 1.0/s, (5, 3), false }: 0	
Current Index	0
Concrete Paths	

Ordered Ends	#1
{ 0, 2.0/s, (0, 6), true }: 1	
{ 0, 1.0/s, (1, 4), false }: 1	
{ 0, 1.0/s, (5, 3), false }: 0	
Current Index	1
Concrete Paths	
End:0 -> End:1, 1.0/s	

Ordered Ends	#2
{ 0, 2.0/s, (0, 6), true }: 1	
{ 0, 1.0/s, (1, 4), false }: 1	
{ 0, 1.0/s, (5, 3), false }: 0	
Current Index	2
Concrete Paths	
End:0 -> End:1, 1.0/s	

Ordered Ends	#3
{ 0, 2.0/s, (0, 6), true }: 1	
{ 0, 1.0/s, (1, 4), false }: 1	
{ 0, 1.0/s, (5, 3), false }: 1	
Current Index	-
Concrete Paths	
End:0 -> End:1, 0.0/s	
Path:0 -> End:2, 0.0/s	

Figure 7: Processing of endpoints to paths, using Figure 5.

In iteration #0, endpoint 0 is paired with endpoint 1 with minimum left over of 1.0/s, however it could've equally also used endpoint 2. The rate left over in the path group is initialized with $source - dest = 1$. The next iteration #1 skips due to endpoint 1 already having been satisfied. Iteration #2 looks at endpoint 2 as a destination requiring 1/s and finds the previously calculated concrete path 0 as a source with 1/s item rate left. This path is added to path group 0 with the previous path, which is updated to have 0/s leftover rate. At this point all the endpoints are satisfied.

Considering the definition of the path configurations, my choice of MAPF algorithm is limited. A path defined with another path at either end cannot trivially be calculated in parallel with the path it is dependent on. Hence, the MAPF pathfinder must have some ordering when calculating paths. It also needs to be able to calculate the exact position the depended upon path is split from when it is needed in the dependant path. The decoupled WHCA* algorithm [15] as defined by D. Silver calculates the paths separately in a specific ordering, each considering the previous. CBS [5] as defined by Sharon et al. instead calculates them simultaneously without any order, and then finds and handles conflicts afterwards. WHCA* can be directly applied to this problem, however CBS requires more thought. It is possible to be applied if you calculate the separate CBS paths in a specific order, as determined by the path dependencies. Furthermore, recalculations of a given path (as done in both) affects all dependant paths, and as such you also need to resolve endpoint positions for each dependant path every time you recalculate a depended upon path. For both algorithms further changes are needed; the reservation table in WHCA* and the

conflicts in the Constraints Tree defined in the CBS do not need to consider the time dimension with their constraint's definition and furthermore a *wait* action has no meaning for these paths. The definition used for comparing conflicting states is domain specific with this problem which will be further discussed later, however both algorithms can support this modified representation (as opposed to the standard (x, y, t)). Considering the above, both algorithms are applicable to this problem with specific tweaking. Sharon et Al [5] find that CBS will always find an optimal solution given their definition, and hence I choose to implement CBS.

3.2.3 Low Level Pathfinding

I will explain modifications needed to be made to allow CBS to function, but first a decision on the low-level pathfinding. When given a concrete source and destination position, standard A* [6] is adequate to be able to path find using a domain specific state representation. The A* search will find the path through the state space induced by the neighbourhoods of each state, minimising the cost to the goal.

I chose for my state to represent a position, a direction, and a belt type. The belt type can be *Conveyor*, *Underground*, *Underground Entrance, Underground Exit, Inserter*, and *None*. The *Inserter* and *None* types are only used for initial nodes and therefore are not reachable from the neighbourhood of any other state, however they do each define a neighbourhood. *Inserter* is used when an initial node has been branched off another path as such with a path configuration that resolves a path as a source. *None* is used as an initial node starting from a defined source item endpoint. All the states calculated must have positions within the bounds of the blueprint, and unless otherwise stated cannot be in a position that is marked in the blocked grid.

The neighbourhood of a *Conveyor* is straightforward: it can go forward, left, or right, based on the state's direction. These can be either *Conveyor* or *Underground Entrance* states at each position. *Underground* can only go forward; this holds true for the *Underground Entrance* and *Underground Exit* also. *Underground Entrance can only go to Underground. Underground* can go to *Underground* or *Underground Exit*. *Underground Exit* can go to *Underground Entrance* or *Conveyor*. *Inserters* are defined the same as *Underground Exit*, as only being able to go forward, and either *Entrance* or *Conveyor*. *None* is a special case state type, in that they can go to *Conveyor* or *Entrance* in the *same* position, in any direction. Equality comparison (for the sake of the open and closed set) requires the states position, type, and direction to all be the same.

Due to including types and directions for the pathfinding states, it would be too strict to define a specific goal state that must be directly

met. Instead, I define a pathfinding goal as a position, a list of types, and a list of directions. To calculate the heuristic distance, I use the Manhattan Distance between the positions. To check if a state has reached the goal, we ensure that the positions match, and then if either of the goal lists are non-empty then ensure that the state is within the list. Otherwise, I match any state with an empty list. For example, when pathing to an Item Endpoint we want the goal to be at the same position, one of the types of *Conveyor, UndergroundEntrance, UndergroundExit*, and in any direction. When going to a path however we need to be either *Conveyor* or *UndergroundExit* and directed towards the path (so to move items onto the conveyor).

When calculating the distance between states, generally I calculate the Manhattan Distance between the positions of the states. In this problem however I want to reward underground belt usage; it is intuitive that because underground belts block less positions within the blueprint, it will likely produce a more optimal result. To encode this, I check if the 2 nodes are both *Underground*, and if so halving the calculated distance. This is performed for the distance from a node to its parent, when counting the cumulative cost within A*.

With using CBS as a higher level I need some concept of a conflict between states. Due to this unique representation, this isn't trivial. If 2 states are in the same position and both are above ground, e.g. *Conveyor, Inserter, Underground Entrance / Exit* then they will conflict. Similarly, if they are both in the same position, underground, and specifically are oriented the same axis (e.g. both vertical going either North or South, or both horizontal going either West or East) they will collide. Complexity comes here, in that the *Underground Entrance / Exit* have both above ground and underground presence. To implement this efficiently, I represent this presence in a 3-bit flag, in which bit 0 represents above ground, bit 1 represents below ground vertical, and bit 2 represents below ground horizontal. Each state may have 0 or more of these toggled. For example, a *Conveyor* is 0b001, an *Underground* going North is 0b010, an *Underground Entrance* going East is 0b101, *None* is 0b000. Then to check for a conflict, first I check the positions are equal, and then using bitwise arithmetic I check $(conflictFlags \,\&\, other.conflictFlags) > 0$. This effectively checks whether any of the flags are true in both states. My pathfinding state can then be reduced to 2 integers: a hash of the position and a 3-bit integer containing the conflict flags. This can be used as an entry into the CBS CAT or used to define the constraints imposed on a path.

When the CBS has a CT node with a path to recalculate, it will first look at the path dependencies (as calculated when the path configurations were calculated) and order the path configurations accordingly, such that depended upon paths are calculated first. Then, for each path configuration to calculate with its set of constraints, I first

need to resolve it into a source pathfinding state and a goal for A*. With an item endpoint source or destination this is straight forward. For a source, I produce a state with the same position, the *None* type, and arbitrarily the North direction. For a goal I use the destination position, a list of any above ground types (*Conveyor, UndergroundEntrance, UndergroundExit*), and allow any. I also make sure to check that the source position / type / direction does not violate any of the path's constraints nor does it conflict with any entries in the CAT. When using a path as a source or destination however this is more complex. As I know the path has already been calculated, I loop over the nodes and check each of the 4 adjacent positions. For each, I check that it is within the bounds of the blueprint, not blocked by the blocked grid, does not have any entries into the CAT (and therefore does not overlap existing paths), and does not conflict with any constraints of the path I am resolving. For each of these valid positions, I find the one closest to the other endpoint. With a path as a source, the source state will use this position, the *Inserter* type, and the direction facing away from the path. In this case specifically, I also perform the above checks on the square 1 further space away from the path, as to check that the inserter I am placing has somewhere to go. With a path as a destination, the goal is the same position, either *Conveyor* or *ConveyorExit*, and facing towards the path as to be moving items onto it. If the CBS is unable to resolve a configuration (e.g. a source or destination is blocked) then the CT node is discarded.

With all this defined, the general procedure for calculating the fitness for a given Simulated Annealing state can be laid out. First calculate the blocked grid and the item endpoints, to find the overlap penalty. Assuming this is 0, perform the disjoint endpoint algorithm over the given item endpoints to find concrete path configurations. CBS is then performed using these as input alongside the blocked grid. When a CT node is calculating a path it queues up to also calculate any dependant paths, resolves the configurations to a concrete A* source and goal, and then performs the A* algorithm with the given path's constraints. Once CBS finds a valid series of paths for the current local search state, the pathfinding fitness can be calculated. The pathfinding fitness is found as a sum over all the successfully calculated paths: $1 + \min(1, \frac{euclideanDistance}{calculatedDistance})$. I ensure every found path contributes a minimum fitness of 1, however I also want to reward more efficient paths. Generally, a paths minimum possible distance would be the Manhattan Distance, however considering underground belts it is possible to go below this, and so I constrain the additional reward to 1. Therefore, each path will contribute between 1 and 2 fitness. The final fitness of a local search state is therefore $pathfindingFitness - overlapPenalty$.

4 Results and Discussion

4.1 Results

To evaluate my design and implementation I will pose 3 example problems as seen in *figure 8*.

- **Problem 1**: Minimal 5x5 blueprint, with a single input / output item, and intended to be easy to solve.
- **Problem 2:** More standard scenario in a 10x10 space with a more realistic set of recipes.
- **Problem 3:** Larger at 15x15 and intended for comparing different run configurations.

The run configurations for each problem can be found in **Appendix A**.

Problem 1			
Blueprint Size: 5x5			
Recipes			
1 **(0)**	→	1 **(1)**	1s
Inputs / Outputs			
Input: Item **(0)** at **(0, 1)** with rate **1/s**			
Output: Item **(1)** at **(4, 2)**			

Problem 2			
Blueprint Size: 10x10			
Recipes			
1 **(0)**	→	1 **(1)**	0.5s
2 **(0)** + 2 **(1)**	→	1 **(2)**	0.5s
Inputs / Outputs			
Input: Item **(0)** at **(0, 1)** with rate **4/s**			
Output: Item **(2)** at **(9, 9)**			

Problem 3			
Blueprint Size: 15x15			
Recipes			
1 **(0)** + 2 **(1)**	→	1 **(2)**	2s
3 **(0)** + 1 **(2)**	→	1 **(3)**	0.5s
Inputs / Outputs			
Input: Item **(0)** at **(0, 2)** with rate **2/s**			
Input: Item **(1)** at **(0, 10)** with rate **4/s**			
Output: Item **(3)** at **(14, 14)**			

Figure 8: 3 example problems for evaluation.

I first evaluated the effectiveness of Simulated Annealing as a top-level local search for minimising overlapping of the layout against a baseline of Hill Climbing. I disabled the CBS pathfinding fitness component for each state ran the solver 5 times, each time with a different random starting state, and tracking the overlap penalty over time. For both algorithms I use a maximum of 1000 iterations. With simulated annealing I chose parameters of initial temperature = 2 and cooling rate = 0.005 based on preliminary testing, with the intention of maximizing exploration at the start, and allowing it to settle before the end. *Figure 9* shows the best run for each task.

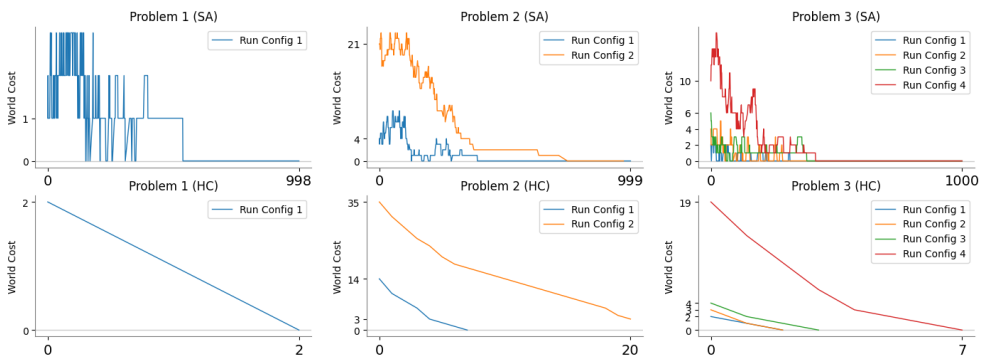

Figure 9: Local search algorithm evaluation graph.

Both algorithms optimized a solution with 0 overlap penalty for each of the 3 problems. *Table 1* contains the number of 0 overlap penalty solutions found for each task and highlights how SA was consistently more successful across the board. HC did always find its solution in considerably fewer iterations; however, this may be due to the parameter chosen for SA. These overall results suggest that a Hill Climber with restarts (e.g. restarting multiple times and picking best fitness) for overlap penalty minimisation can perform with similar effectiveness but in less iterations than SA.

	Run Config 1	Run Config 2	Run Config 3	Run Config 4
Problem 1, HC	3			
Problem 1, SA	4			
Problem 2, HC	3	0		
Problem 2, SA	5	1		
Problem 3, HC	5	5	5	5
Problem 3, SA	5	5	5	5

Table 1: Local search algorithm evaluation results.

Reintroducing CBS to the local state fitness calculation, I evaluated the full solver against the same tasks. Each top-level local search still uses the same parameters as before and are still ran 5 times each. *Figure 10* shows the fitness from the best run for each task.

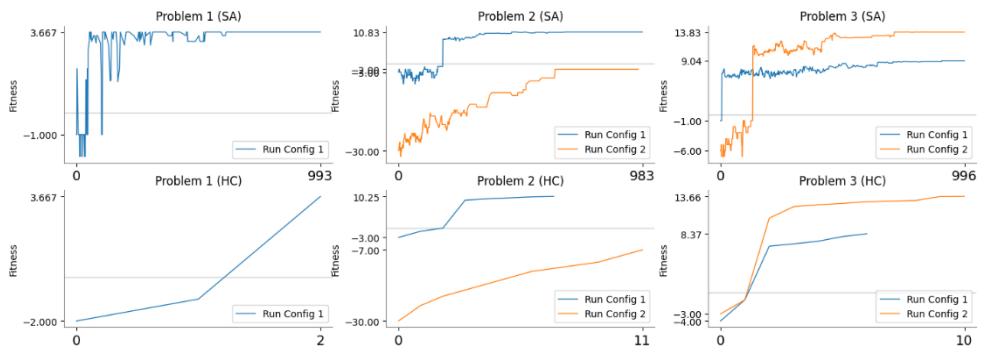

Figure 10: Full solver evaluation graph.

Both top level search algorithms found a solution for each problem with all item endpoints satisfied. Both did however fail to find a solution for problem 2 run configuration 2 within these 5 specific runs, despite a solution found in the previous results (*Table 1*). During gathering of these results, an issue with the solution was highlighted; In problems with a larger blueprint size and larger pathfinding search space (such as problem 3 run configuration 3 / 4) the CBS search space grows drastically, causing the computation to be excessively slow. As such, these 2 tasks were not able to be evaluated for the above results. Further optimization will be discussed in **Section 4.2**.

To better compare the differences between the 2 methods, *Figure 11* shows each of the 5 runs for each task in problem 2. As before with the overlap penalty evaluation, SA managed to find a solution and achieve higher fitness more consistently across the board. For run configuration 2, HC seemed to struggle getting past a threshold of overlap penalty 7, while SA was able to escape this to a minimum of overlap penalty 2. This highlights a key reason for picking SA; its ability to escape local minima. As per the results seen in *Table 1*, it is possible to achieve an overlap penalty of 0 for run configuration 2, however this was not reliably achieved with the current SA parameters.

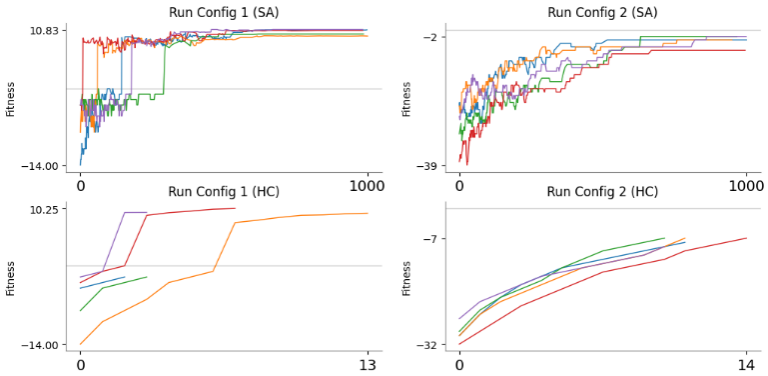

Figure 11: Problem 2 evaluation graphs.

Figure 12 shows the problem definition for problem 2, and *Figures 12 a-c* show some optimized solutions. These have the following fitnesses: $A: 10.8$ | $B: 10.83$ | $C: 10.20$.

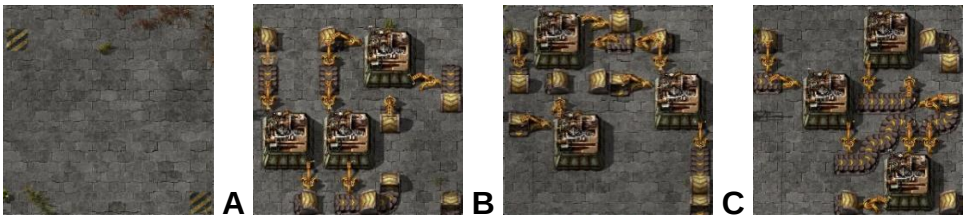

Figure 12, 12a-c: Problem 2 definition and examples.

Visualising these final solutions exposes a key oversight, as outlined in *Figures 13a* and *13b.* The path configuration resolution directs paths into the sides of underground entrances / exits when connecting onto other paths. This does not consider that Items cannot be transferred onto the non-conveyor edges, and in turn would be invalid in game solutions. Potential solutions will be discussed in **Section 4.2**.

Figures 13a, 13b: Highlighted problems with path resolution.

4.2 Future Work

Considering the results, there are a series of possible improvements to the design. It may be more efficient to use a Hill Climber until a solution with 0 overlap penalty is found, then swapping to Simulated Annealing. The HC achieves the same fitness in less iterations, and therefore this may cut down on computation time. Furthermore, there may be space to optimize the CBS. For example, considering more than 2 paths at each conflict and branching the CT accordingly may help explore the CBS search space faster. Also, with some form of hashing of a CT node you could prevent recalculation in case the same node is visited twice through memoization. For improving the quality of the A* pathfinding solutions you could pass the CBS CAT down and use it to break ties between nodes with equal fitness, as described by Sharon et al. [5].

The key issue noted at the end of the evaluation section requires more consideration. The path resolution allows the ends of paths to go into the side of Conveyors, Underground Entrances, and Underground Exits. This caused an issue when verified in-game, due to the underground entrance / exits not being able to receive items on all their sides. Changing this goal state to be strictly just conveyors causes issues, as due to the preference for underground belts the majority of each path is underground entrance / exits, and therefore other paths would have nowhere to path find onto. Potential ways to solve this may be through a heuristic within each path that considers dependant paths, or alternatively, when calculating a path and its dependant paths during a CT node you could instead performing these in a coupled A* search, calculating each path group at the same time.

The layout optimization is still based on an abstract model of the game and therefore there are still features that should be reintroduced for a complete optimization. Electricity is omitted; machines in-game are required to be within a defined radius of an electricity pole, and these poles connect to each other within another larger radius. If a solution is optimized too compactly, then there may be no space for these poles. Reid. et Al. suggest forms of co-evolution [10], however where this may fit within this model is non-trivial. Furthermore, conveyors in Factorio have 2 separate lanes, which I abstract here to a single combined flow rate. Real optimal solutions may consider using each side of a conveyor for different items, which is not done here. Conveyor splitters / mergers are also not modelled and can allow for more precise flow rate control.

4.3 Summary

Overall, I was successful in achieving the goals I initially set out for the project. I was able to apply local search for optimal machine placement and was capable of composing this with an implementation of CBS MAPF for routing conveyors within the bounds Factorio defines. The final implementation was capable of solving a series of example problems and finding solutions that were applicable in game. I highlighted a series of places for further improvement, namely slow calculation during the CBS conflict tree traversal, as well as space for further optimization, for example considering for the CAT in the low-level A* search, and better conflict branching when more than 2 conflicts are found in the CBS.

Considering the validity of the solutions when implemented in-game, there were some noted concerns. The exact implementation of the A* search and path resolution led to conveyor scenarios that wouldn't work as expected which I noted would not be trivial to consider and fix. Furthermore, omitting details such as 2 lane conveyors and electricity distribution means the solutions may not work, and likely would not be optimal, therefore there is still work to do before the solutions returned from a solver of this kind would be and preferred over manually produced, more optimal solutions.

Appendix A: Problem Run Configurations

Example problem 1 run configurations.

```
Run Config with output assemblers = 1
        Input item: 0
        Component item: 1
          Assemblers: 1
          Output inserters / assembler: 1 @ 1/s
          Input inserters / assembler for 0: 1 @ 1/s
```

Example problem 2 run configurations.

```
Run Config with output assemblers = 2
        Input item: 0
        Component item: 1
          Assemblers: 4
          Output inserters / assembler: 1 @ 0.5/s
          Input inserters / assembler for 0: 1 @ 0.5/s
        Component item: 2
          Assemblers: 2
          Output inserters / assembler: 1 @ 0.5/s
          Input inserters / assembler for 0: 1 @ 1/s
          Input inserters / assembler for 1: 1 @ 1/s

Run Config with output assemblers = 1
        Input item: 0
        Component item: 1
          Assemblers: 2
          Output inserters / assembler: 1 @ 0.5/s
          Input inserters / assembler for 0: 1 @ 0.5/s
        Component item: 2
          Assemblers: 1
          Output inserters / assembler: 1 @ 0.5/s
          Input inserters / assembler for 0: 1 @ 1/s
          Input inserters / assembler for 1: 1 @ 1/s
```

Example problem 3 run configurations.

```
Run Config with output assemblers = 4
        Input item: 0
        Input item: 1
        Component item: 2
          Assemblers: 1
          Output inserters / assembler: 1 @ 2/s
          Input inserters / assembler for 0: 1 @ 2/s
          Input inserters / assembler for 1: 1 @ 4/s
        Component item: 3
          Assemblers: 4
          Output inserters / assembler: 1 @ 0.5/s
          Input inserters / assembler for 0: 1 @ 1.5/s
          Input inserters / assembler for 2: 1 @ 0.5/s

Run Config with output assemblers = 3
        Input item: 0
        Input item: 1
        Component item: 2
          Assemblers: 1
          Output inserters / assembler: 1 @ 1.5/s
          Input inserters / assembler for 0: 1 @ 1.5/s
          Input inserters / assembler for 1: 1 @ 3/s
        Component item: 3
          Assemblers: 3
          Output inserters / assembler: 1 @ 0.5/s
          Input inserters / assembler for 0: 1 @ 1.5/s
          Input inserters / assembler for 2: 1 @ 0.5/s

Run Config with output assemblers = 2
        Input item: 0
        Input item: 1
        Component item: 2
          Assemblers: 1
          Output inserters / assembler: 1 @ 1/s
          Input inserters / assembler for 0: 1 @ 1/s
          Input inserters / assembler for 1: 1 @ 2/s
        Component item: 3
          Assemblers: 2
          Output inserters / assembler: 1 @ 0.5/s
          Input inserters / assembler for 0: 1 @ 1.5/s
          Input inserters / assembler for 2: 1 @ 0.5/s

Run Config with output assemblers = 1
        Input item: 0
        Input item: 1
        Component item: 2
          Assemblers: 1
          Output inserters / assembler: 1 @ 0.5/s
          Input inserters / assembler for 0: 1 @ 0.5/s
          Input inserters / assembler for 1: 1 @ 1/s
        Component item: 3
          Assemblers: 1
          Output inserters / assembler: 1 @ 0.5/s
          Input inserters / assembler for 0: 1 @ 1.5/s
          Input inserters / assembler for 2: 1 @ 0.5/s
```

Bibliography

[1] “Factorio Steam Page,” Wube Software, [Online]. Available: https://store.steampowered.com/app/427520/Factorio/. [Accessed 30 04 2024].

[2] “Factorio Wiki, Generation,” Wube Software, [Online]. Available: https://wiki.factorio.com/Map_generator. [Accessed 30 4 2024].

[3] S. Patterson, J. Espasa, M. S. Chang and R. Hoffmann, “Towards Automatic Design of Factorio Blueprints,” *arXiv:2310.01505,* 2023.

[4] S. Russel and P. Norvig, “Beyond Classical Search,” in *Artificial Intelligence, A modern Approach, Third Edition*, PEARSON INDIA, 2015, pp. 120-160.

[5] G. Sharon, R. Stern, A. Felner and N. R. Sturtevant, “Conflict-Based Search For Optimal Multi-Agent Path Finding,” *Artificial Intelligence,* vol. 219, pp. 40-66, 2015.

[6] S. Russel and P. Norvig, “A* search: Minimizing the total estimated solution cost,” in *Artificial Intelligence, A modern Approach, Third Edition*, PEARSON INDIA, 2015, pp. 93-99.

[7] X. Peng, C. Solnon and O. Simonin, “Solving the Non-Crossing MAPF with CP,” in *27th International Conference on Principles and Practice of Constraint Programming*, 2021.

[8] “Factorio,” Wube Software, [Online]. Available: https://factorio.com/. [Accessed 30 04 2024].

[9] A. Leue, “Verification of Factorio Belt Balancers using Petri Nets,” Universitats- und Landesbibliothek, 2021.

[10] K. Reid, I. Miralavy, S. Kelly, W. Banzhaf and C. Gondro, “The Factory Must Grow: Automation in Factorio,” *arXiv:2102.04871,* 2021.

[11] M. Lori, V. de Lima, S. martello and F. Miyazawa, “Exact solution techniques for two-dimensional cutting and packing,” *European Journal of Operational Research,* vol. 289, pp. 399-415, 2021.

[12] J. Yu and S. LaValle, “Multi-agent Path Planning and Network Flow,” *Algorithmic Foundations of Robotics X. Springer Tracts in Advanced Robotics,* vol. 86, pp. 157-173, 2012.

[13] I. Whalen, W. Banzhaf, A. Mamun, H.A. and C. Gondro, “Evolving SNP Panels for Genomic Prediction,” in *Evolution in Action: Past, Present and Future*, Michigan, Springer Cham, 2020, pp. 467-487.

[14] L. R. Ford Jr. and D. R. Fulkerson, “Maximal Flow Through a Network,” *Canadian Journal of Mathematics,* vol. 8, pp. 399-404, 1956.

[15] D. Silver, “Cooperative Pathfinding,” *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment,* vol. 1, no. 1, pp. 117-122, 2005.

[16] S. Russel and P. Norvig, “Simulated Annealing,” in *Artificial Intelligence, A modern Approach, Third Edition*, PEARSON INDIA, 2015, p. 125.